

TD Feuille 8 — Propriétés des langages, sûreté et sécurité

Exercice 1. Typages statiques et dynamiques dans les langages orientés objets

```

1 class A:
2     pass
3
4 class B(A):
5     def hello():
6         print("hello")
7
8
9
10 def foo(x):
11     o = A
12     if x % 2 == 0:
13         o = B
14     o.hello()

```

FIGURE 1 – Python

```

class A { }
class B extends A {
    void hello() {
        System.out.println
            ("hello");
    }
}

void foo(int x) {
    A o = new A();
    if (x % 2 == 0) {
        o = new B();
    }
    o.hello();
}

```

FIGURE 2 – Java

```

class A
end
class B < A
    def hello : Nil
        puts("hello")
    end
end
def foo(x : Int32) : Nil
    o : A = A.new
    if x % 2 == 0
        o = B.new
    end
    o.hello
end

```

FIGURE 3 – Crystal

► **Question 1.** En quoi le typage fort dans les langages de programmation PYTHON et JAVA protège-t-il les programmes de vulnérabilités présentes dans les langages sans typage (comme l'assembleur) ou avec un typage faible (comme le C)?

Des langages comme OCAML ont un typage uniquement statique. D'autres comme PYTHON uniquement dynamiques (même si des annotations de types peuvent être données). JAVA, lui dans sa sémantique, combine typage statique et dynamique. Quels sont les avantages et inconvénients comparatifs du typage statique versus dynamique?

► **Question 2.** On considère les programmes des figures 1 et 2. Pour quelles valeurs de x la fonction `foo` en PYTHON s'exécute-t-elle sans erreur? Le code JAVA est rejeté par le compilateur : expliquer pourquoi, sur quelle ligne, et comment modifier le programme pour qu'il soit accepté par le compilateur et se comporte de manière comparable au programme PYTHON.

► **Question 3.** Le langage CRYSTAL¹ est un langage orienté objet qui combine un typage statique et du typage dynamique, un peu comme JAVA. Par exemple, le programme à la figure 3 est rejeté à compilation : pour le faire accepter, il faut introduire un cast explicite à la ligne 14 : `o.as(B).hello`. Par contre, il a un typage statique basé sur une analyse *par flot de données* qui permet au type statique d'une variable de changer d'un point de contrôle à l'autre. Par exemple, la variante de l'exemple donnée à la figure 5 est acceptée par le compilateur. Est-ce le cas en Java pour la figure 4? Si non, que doit faire le programmeur pour faire accepter le programme? Est-ce possible en ne faisant aucun cast explicite?

1. CRYSTAL a une syntaxe inspirée du très populaire langage RUBY, mais avec un typage statique qui favorise une compilation en code natif efficace – via LLVM. Sa première release date de 2014. Voir [https://en.wikipedia.org/wiki/Crystal_\(programming_language\)](https://en.wikipedia.org/wiki/Crystal_(programming_language)) et <https://crystal-lang.org/reference>.

```

1 A foo(int x) {
2   A o = new A();
3   if (x % 2 == 0) {
4     o = new B();
5     o.hello();
6   }
7   return o;
8 }

```

FIGURE 4 – Java

```

1 def foo(x : Int32) : A
2   o : A = A.new
3   if x % 2 == 0
4     o = B.new
5     o.hello
6   end
7   o
8 end

```

FIGURE 5 – Crystal

Exercice 2. Vérifications statiques et dynamiques du dérèfencement de l'objet nul

```

void bar(B o) {
  o.hello();
}

```

FIGURE 6 – Java

```

def bar(o : B) : Nil
  o.hello
end

```

FIGURE 7 – Crystal

```

def bar(o : B | Nil) : Nil
  if o.is_a?(Nil)
    puts("oops: sorry")
  else
    o.hello
  end
end

```

FIGURE 9 – Crystal

```

1 def foo(x : Int32) : Nil
2   o : B | Nil = nil
3   y : Int32 = x + x
4   if y % 2 == 0
5     o = B.new
6   end
7   o.hello
8 end

```

FIGURE 8 – Crystal

```

def bar(o : B | Nil) : Nil
  if o.is_a?(Nil)
    o = B.new
  end
  o.hello
end

```

FIGURE 10 – Crystal

► **Question 1.** En JAVA, l'objet `null` est implicitement d'un sous-type de n'importe quel type (cf. projet GL) : son type est donc le minimum dans le treillis des types objets (Object étant le maximum). Sur le code de la figure 6, que se passe-t-il pour un appel `bar(null)` ; ? Est-ce accepté à la compilation : si non, quel est le message d'erreur ; si oui, quel est le comportement à l'exécution ?

► **Question 2.** En CRYSTAL, il y a un objet `nil` qui est l'unique habitant du type `Nil`. Mais ce type n'est pas un sous-type de n'importe quel type. Sur le code de la figure 7, un appel à `bar(nil)` est rejeté statiquement. On peut laisser la possibilité aux objets de prendre le type `Nil` en utilisant l'opérateur `|` qui représente une "union" (binaire) sur les types, comme aux figures 9 et 10. Pour tester qu'un objet `o` a un type *dynamique* qui est un sous-type de `A`, on utilise la construction `o.is_a?(A)` qui a la même sémantique à l'exécution que "`o instanceof A`" en JAVA. En CRYSTAL, cette construction a aussi un effet sur le typage statique, qui permet par exemple d'accepter les programmes des figures 9 et 10.

Expliquer *pourquoi* le compilateur CRYSTAL peut garantir que dans les deux programmes des figures 9 et 10, la ligne `o.hello` ne va jamais provoquer d'erreur à l'exécution. Quel est à votre avis l'effet de `is_a?` sur le typage statique pour que le compilateur réussisse à détecter que ces programmes sont sans erreur à l'exécution ?

▸ **Question 3.** Le programme de la figure 8 n'est pas accepté par le compilateur Crystal. Ce programme présente-t-il un risque d'erreur à l'exécution? Si non, pourquoi le compilateur rejette-t-il le programme? Comment le corriger pour le faire accepter? Conclure en indiquant, à votre avis, quels sont les avantages et les inconvénients du traitement de l'objet `nil` en CRYSTAL.

Exercice 3. Protection contre les Use-After-Free à la Rust

Dans cet exercice, on s’inspire du langage RUST pour se protéger des “Use-After-Free” (UAF) en C.² RUST utilise la notion “d’ownership” qui garantit que toute valeur allouée dynamiquement a un unique possédant qui gère sa durée de vie. L’ownership peut être transféré lors de l’affectation ou du passage de paramètres. Dans cet exercice on va chercher à simuler cette notion en C.

On considère le programme donné en figure 11 qui manipule des tableaux alloués dynamiquement et pour lesquels on dispose de la taille (comme en Java). Les fonctions `new_array`, `access`, `resize_array` et `free_array` permettent respectivement d’allouer un tableau, d’accéder à un élément du tableau, de redimensionner un tableau et de le désallouer (“drop” dans la terminologie RUST). Si besoin ces fonctions sont définies dans la figure 12.

```

1  /*** Un type abstrait de tableaux redimensionnables ***/
2  typedef struct _array /*move*/ *array;
3  array new_array(size_t len);           // allocation
4  int *access(array *a, size_t i);      // accès à la case i
5  void resize_array(array *a, size_t len); // redimensionnement
6  void free_array(array a);             // désallocation
7
8  /*** "Untrusted" client ***/
9  void main(int argc) {
10     array a = new_array(10);
11
12     for (int i=0; i<10; i++) *access(&a,i) = i;
13     resize_array(&a, 20);
14     for (int i=0; i<10; i++) *access(&a,i+10) = *access(&a,i)*10;
15     for (int i=0; i<20; i++) printf("%d\n", *access(&a,i));
16
17     free_array(a);
18
19     printf("argc = %d\n", argc);
20     switch (argc) {
21     case 2: printf("%d\n", *access(&a,1)); break;
22     case 3: free_array(a); break;
23     case 4: a = new_array(5); break;
24     case 5: resize_array(&a, 5); break;
25     }
26 }

```

FIGURE 11 – Un client avec potentiellement des bugs

► **Question 1.** Pour chaque accès dans le client (lignes 9 à 26 de la figure 11) indiquer s’il y a des UAFs.

► **Question 2.** Concrètement, le programmeur C déclare certains types de pointeurs avec un “move” en commentaire (comme dans l’exemple pour le type `array`, ligne 2 Fig. 11). Cela signifie que la copie des variables des types pointeurs annotés par “move” (dans les affectations ou en passage de paramètre) est vue comme *destructive* : la variable “copiée”

2. Mais il n’est pas nécessaire de connaître RUST pour comprendre/faire cet exercice.

```

1  /*** implémentation du type abstrait "array" ***/
2  struct _array { size_t len; int /*move*/ *contents; };
3
4  array new_array(size_t len) {
5      array a = malloc(sizeof(struct _array));
6      if (a == NULL) abort();
7      a->contents=malloc(len*sizeof(int));
8      if (a->contents == NULL) abort();
9      a->len=len;
10     return a;
11 }
12
13 int *access(array *a, size_t i) {
14     if (i >= (*a)->len) abort();
15     return &((*a)->contents[i]);
16 }
17
18 void resize_array(array *a, size_t len) {
19     (*a)->contents = realloc((*a)->contents, len*sizeof(int));
20     if ((*a)->contents==NULL) abort();
21     (*a)->len=len;
22 }
23
24 void free_array(array a) {
25     free(a->contents);
26     free(a);
27 }

```

FIGURE 12 – Type abstrait des tableaux redimensionnables

devient inutilisable après la copie. Pour copier ces pointeurs sans les détruire, le programmeur s’astreint à les “*emprunter*” par adresse mais s’interdit alors d’utiliser ces emprunts pour détruire la donnée : pour emprunter un pointeur “*a*”, il écrit “*&a*” et manipule donc un pointeur de pointeur au lieu d’un simple pointeur.³

Indiquer pour chaque utilisation de la variable “*a*” dans le client, si c’est un “*move*” ou un “*emprunt*”. Y a-t-il d’autres cas d’utilisation de la variable “*a*” à considérer dans cet exemple?

► **Question 3.** Proposer une analyse inspirée des analyses de bonne initialisation (cf. exo 2 de la feuille 7) pour détecter les variables accédées après un “*move*”. Comment se comporte votre analyse sur un code de la forme “*free_array(a); a = new_array(5);*”?

► **Question 4.** Quelles sont les erreurs détectées par cette analyse dans le “*main*” de la figure 11?

► **Question 5.** Dans le cas général, cette analyse a-t-elle des faux positifs (signale des UAF qui n’en sont pas)? Justifier (avec un contre-exemple ou l’esquisse d’une preuve).

3. Dans cet exercice, on n’utilise qu’une seule forme d’emprunt, i.e. sans distinguer les emprunts “*mutables*” des emprunts “*partageables*” comme fait Rust.

► **Question 6.** On considère maintenant l'exemple ci-dessous :

```
1   array *emprunt = &a;  
2   free_array(a);  
3   *access(emprunt, 0) = 7;
```

Expliquer en quoi c'est un exemple de faux négatif (UAF non vu) pour votre analyse? Comment pourrait-on les éviter?

► **Question 7.** Certains langages comme PYTHON ou JAVA interdisent par construction les UAFs. Si ce n'est pas le cas on peut imaginer une analyse statique détectant les UAF ou bien leur détection à l'exécution comme vu en TP (e.g. option `-fsanitize=address`). Discuter de ces différents choix vis-à-vis du développeur, des assurances en sécurité et de la performance.